RUN-TIME PARALLELIZATION OF LOOPS IN COMPUTER PROGRAMS

Field of the invention

The present invention relates to run-time parallelization of computer programs that have loops containing indirect loop index variables and embedded conditional statements.

Background

A key aspect of parallel computing is the ability to exploit parallelism in one or more loops in computer programs. Loops that do not have cross-iteration dependencies, or where such dependencies are linear with respect to the loop index variables, one can use various existing techniques to achieve parallel processing. A suitable reference for such techniques is Wolfe, M., *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996, Chapters 1 and 7. Such techniques perform a static analysis of the loop at compile-time. The compiler suitably groups and schedules loop iterations in parallel batches without violating the original semantics of the loop.

There are, however, many cases in which static analysis of the loop is not possible. Compilers, in such cases, cannot attempt any parallelization of the loop before run-time.

As an example, consider the loop of **Table 1** below, for which parallelization cannot be performed.

TABLE 1

```
do i = 1, n
  x[u(i)] = . . . .
  . . . .   . . . .
  . . . .   . . . .
  y[i] = x[r(i)] . . .
  . . . .   . . . .
  . . . .   . . . .
enddo
```

Specifically, until the indirect loop index variables **u(i)** and **r(i)** are known, loop parallelization cannot be attempted for the loop of **Table** 1.

For a review on run-time parallelization techniques, refer to Rauchwerger, L., *Run-Time Parallelization: It's Time Has Come*, Journal of Parallel Computing, Special Issue on Language and Compilers, Vol. 24, Nos. 3-4, 1998, pp. 527-556. A preprint of this reference is available via the World Wide Web at the address *www.cs.tamu.edu/faculty/rwerger/pubs.*

Further difficulties, not discussed by *Wolfe* or *Rauchwerger*, arise when the loop body contains one or more conditional statements whose evaluation is possible only during runtime. As an example, consider the loop of **Table 2** below, for which parallelization cannot be attempted by a compiler.

TABLE 2

```
do i = 1, n
  x[u(i)] = . . . .
  . . . .   . . . .
  . . . .   . . . .
if (cond) then y[i] = x[r(i)] . . .
else y[i] = x[s(i)] . . .
  . . . .   . . . .
  . . . .   . . . .
enddo
```

The value of **r(i)** and **s(i)** in the loop of **Table 2** above, as well as the indirect loop index variables **u(i)** must be known before loop parallelization can be attempted. Further, in each iteration, the value of **cond** must be known to decide whether **r(i)** or **s(i)** should be included in a particular iteration.

Further advances in loop parallelisation are clearly needed in view of these and other observations.

Summary

A determination is made whether a particular loop in a computer program can be parallelized. If parallelization is possible, a suitable strategy for parallelization is provided. The techniques described herein are suitable for loops in which

(i) there are any finite number of array variables in the loop body, such as **x** and **y** in the example of **Table 2** above;

(ii) there are any finite number of indirect loop index variables, such as **u**, **r**, and **s** in the example of **Table 2** above;

(iii) each element of each array variable and of each indirect loop index variable is uniquely identifiable by a direct loop index variable, such as **i** in the example of **Table 2** above;

(iv) the loop index variables (either direct or indirect variables) are not redefined within the loop; and

(v) there are no cross-iteration dependencies in the loop other than through the indirect loop index variables.

Parallelization is attempted at run-time for loops, as noted above, having indirect loop index variables and embedded conditional statements in the loop body. A set of active array variables and a set of indirect loop index variables are determined for the loop under consideration. Respective ranges of the direct loop index values and indirect loop index values are determined. Indirect loop index values are determined for each iteration, and each such value so determined is associated with a unique number. Based on these unique numbers, an indirectly indexed access pattern for each iteration in the loop is calculated.

Using the indirectly indexed access pattern, the loop iterations are grouped into a minimum number of waves such that the iterations comprising a wave have no cross-iteration dependencies among themselves. The waves are then scheduled in a

predetermined sequence and the iterations in a wave are executed independent of each other in the presence of multiple computing processors.

Description of drawings

Fig. 1 is a flow chart of steps involved in performing run-time parallelization of a loop that has indirect loop index variables and one embedded Boolean condition.

Figs. 2A, **2B** and **2C** jointly form a flow chart of steps representing an algorithm for performing run-time parallelization.

Fig. 3 is a schematic representation of a computer system suitable for performing the run-time parallelization techniques described herein.

Detailed description

The following two brief examples are provided to illustrate cases in which an apparently unparallelizable loop can be parallelized by modifying the code, but not its semantics. **Table 3** below provides a first brief example.

TABLE 3

```
b = b0
do i = 1, n
  x[u(i)] = b
  b = b+1
enddo
```

The loop of **Table 3** above cannot be parallelized, since the calculated value of `b` depends on the iteration count `i`. For example, for the 3rd iteration, `x[u(3)] = b0 + 2`, where `b0` is the value of `b` just prior to entering the loop. The loop can, however, be parallelized if the loop is rewritten as shown in **Table 4** below.

TABLE 4

```
b = b0
do i = 1, n
  x[u(i)] = b0 + i - 1
enddo
b = b0 + n
```

Table 5 below provides a second brief example.

TABLE 5

```
do i = 1, n
  c = x[u(i)]
  ... ... ...
enddo
```

The loop of **Table 5** above is parallelizable if the loop is rewritten as shown in **Table 6** below.

TABLE 6

```
do i = 1, n
  c = x[u(i)]
  ... ... ...
enddo
c = x[u(n)]
```

These and other existing rules that improve parallelization of loops can be invoked whenever applicable. The above-mentioned references of Wolfe and Rauchwerger are

suitable references for further such rules that can be adopted as required. The above referenced content of these references is incorporated herein by reference.

Loop parallelization procedure

The loop parallelization procedure described herein is described in greater detail with reference to the example of **Table 7** below.

TABLE 7

```
do i = 5, 15
  x1[r(i)] = s1[u(i)]
  x2[t(i)] = s2[r(i)] * s1[t(i)] . . .
  x3[u(i)] = x1[r(i)]/x3[u(i)]

  if (x2[t(i)]) then x4[v(i)] = s2[r(i)] + x5[t(i)] . . .
  else x3[v(i)] = x5[w(i)]

  x5[u(i)] = x3[v(i)] + x4[v(i)] . . .
  x6[u(i)] = x6[u(i)] - . . .
  x7[v(i)] = x7[v(i)] + x1[r(i)] - s1[u(i)]
  . . . .   . . . .
  . . . .   . . . .
enddo
```

In some cases, the analysis of cross-iteration dependencies is simplified if an array element that appears on the right hand side of an assignment statement is replaced by the most recent expression defining that element, if the expression exists in a statement prior to this assignment statement. In the example of **Table 7** above, `x1[r(i)]` is such an element whose appearance on the right hand side of assignment statements for `x3[u(i)]` and `x7[v(i)]` can be replaced by `s1[u(i)]` since there is an earlier assignment statement `x1[r(i)] = s1[u(i)]`.

Thus, for the example of **Table 7** above, the code fragment of **Table 8** below represents the example of **Table 7** above, after such operations are performed, and represents the results of appropriate replacement.

TABLE 8

```
do i = 5, 15
  x1[r(i)] = s1[u(i)]                    // Defines x1[r(i)]
  x2[t(i)] = s2[r(i)] * s1[t(i)] . . .
  x3[u(i)] = (s1[u(i)])/x3[u(i)]         // Replaces x1[r(i)]

  if (x2[t(i)]) then x4[v(i)] = s2[r(i)] + x5[t(i)] . . .
  else x3[v(i)] = x5[w(i)]

  x5[u(i)] = x3[v(i)] + x4[v(i)] . . .
  x6[u(i)] = x6[u(i)] - . . .
  x7[v(i)] = x7[v(i)]  // Identity after replacing x1[r(i)]
  . . . .   . . . .
  . . . .   . . . .
enddo
```

Further simplification of the code fragment of **Table 8** above is possible if statements that are identities, or become identities after the replacement operations, are deleted. Finally, if the array variable **x1** is a temporary variable that is not used after the loop is completely executed, then the assignment statement defining this variable (the first underlined statement in the code fragment of **Table 8** above) is deleted without any semantic loss, consequently producing the corresponding code fragment of **Table 9** below.

TABLE 9

```
do i = 5, 15
  x2[t(i)] = s2[r(i)] * s1[t(i)] . . .
  x3[u(i)] = (s1[u(i)])/x3[u(i)]

  if (x2[t(i)]) then x4[v(i)] = s2[r(i)] + x5[t(i)] . . .
  else x3[v(i)] = x5[w(i)]

  x5[u(i)] = x3[v(i)] + x4[v(i)] . . .
  x6[u(i)] = x6[u(i)] - . . .
  . . . .   . . . .
  . . . .   . . . .
enddo
```

The array element replacement operations described above with reference to the resulting code fragment of **Table 9** above can be performed in source code, using character string "*find and replace*" operations. To ensure semantic correctness, the replacement string is enclosed in parentheses, as is done in **Table 8** for the example of **Table 7**. To determine if an assignment statement expresses an identity, or to simplify the assignment statement, one may use any suitable technique. One reference describing suitable techniques is commonly assigned United States Patent Application No 09/597,478, filed June 20, 2000, naming as inventor Rajendra K Bera and entitled "*Determining the equivalence of two algebraic expressions*". The content of this reference is hereby incorporated by reference.

Potential advantages gained by the techniques described above are a reduced number of array variables for analysis, and a clearer indication of cross-iteration dependencies within a loop. Further, a few general observations can be made with reference to the example of **Table 7** above.

First, non-conditional statements in the loop body that do not contain any array variables do not constrain parallelization, since an assumption is made that cross-iteration dependencies do not exist due to such statements. If such statements exist, however, a further assumption is made that these statements can be handled, so as to allow parallelization.

Secondly, only array variables that are defined (that is, appear on the left hand side of an assignment statement) in the loop body affect parallelization. In the case of **Table 9** above, the set of such variables, referred to as active array variables, is {**x2**, **x3**, **x4**, **x5**, **x6**} when the condition part in the statement `if (x2[t(i)])` evaluates to *true* and {**x2**, **x3**, **x5**, **x6**} when this statement evaluates to *false*.

If, for a loop, every possible set of active array variables is empty, then that loop is completely parallelizable.

Since detection of variables that affect loop parallelization can be performed by a compiler through static analysis, this analysis can be performed by the compiler. Thus, respective lists of array variables that affect parallelization for each loop in the computer program can be provided by the compiler to the run-time system.

In the subsequent analysis, only indirect loop index variables associated with active array variables are considered. In the example of **Table 9** above, these indirect loop index variables are {**t**, **u**, **v**} when the statement `if (x2[t(i)])` evaluates to *true* and {**t**, **u**, **v**, **w**} when this statement evaluates to *false*.

Let $V \equiv \{v_1, v_2, \ldots v_n\}$ be the set of all active array variables that appear in the loop body, V_T be the subset of V that contains only those active array variables that are active when the Boolean condition evaluates to *true*, and V_F be the subset of V that contains only those active array variables that are active when the Boolean condition evaluates to *false*. Furthermore, let $I \equiv \{i_1, i_2, \ldots i_r\}$ be the set of indirect loop index variables that is associated with the active array variables in V, I_T be the set of indirect loop index variables that is associated with the active array variables in V_T, and I_F be the set of indirect loop index variables that is associated with the active array variables in V_F. Note

that $V \equiv V_T \cup V_F$, $I \equiv I_T \cup I_F$, and the active array variables in $V_T \cap V_F$ are active in the loop body, independent of how the Boolean condition evaluates.

In the example of **Table 9**, these sets are outlined as follows.

$V = \{\mathbf{x2, x3, x4, x5, x6}\}$

$V_T = \{\mathbf{x2, x3, x4, x5, x6}\}$

$V_F = \{\mathbf{x2, x3, x5, x6}\}$

$I = \{\mathbf{t, u, v, w}\}$

$I_T = \{\mathbf{t, u, v}\}$

$I_F = \{\mathbf{t, u, v, w}\}$

Let the values of loop index i range from N_1 to N_2, and those of i_1, i_2, ... i_r range at most from M_1 to M_2.

In the kth iteration (that is, $i = N_1 + k - 1$), the indirect loop index variables have values given by $i_1(i)$, $i_2(i)$, ... $i_r(i)$, and each such value is in the range $[M_1, M_2]$. To facilitate the description of further calculation steps, a different prime number $p(l)$ is associated with each number l in the range $[M_1, M_2]$. The role of these prime numbers is explained in further detail below.

The parallelization algorithm proceeds according to the steps listed below as follows.

1. Create the arrays S_A, S_T and S_F whose respective ith element is given as follows.

   $S_A(i) = \prod_{q \in I} p(q(i))$

   $S_T(i) = \prod_{q \in I_T} p(q(i))$

   $S_F(i) = \prod_{q \in I_F} p(q(i))$

   These array elements are collectively referred to as the *indirectly indexed access pattern* for iteration i. The use of prime numbers in place of the indirect loop index values allows a group of such index values to be represented by a unique number. Thus $S_\alpha(i) = S_\beta(j)$, where $\alpha, \beta \in \{A, T, F\}$, if and only if $S_\alpha(i)$ and $S_\beta(j)$ each contain the same mix of prime numbers. This property follows from the fundamental theorem of arithmetic, which states that every whole number greater than one can be written as

a product of prime numbers. Apart from the order of these prime number factors, there is only one such way to represent each whole number as a product of prime numbers. Note that one is not a prime number, and that two is the only even number that is a prime.

Consequently, if the greatest common divisor (GCD) of $S_\alpha(i)$ and $S_\beta(j)$, is equal to one, there are no common prime numbers between $S_\alpha(i)$ and $S_\beta(j)$, and therefore, no common index values between the ith (α-branch) and the jth (β-branch) iterations. On the other hand, a greatest common divisor greater than one implies that there is at least one common prime number between $S_\alpha(i)$ and $S_\beta(j)$ and, consequently, at least one common index value between the ith (α-branch) and the jth (β-branch) iterations.

The significance of the above result is that if the greatest common divisor of $S_\alpha(i)$ and $S_\beta(j)$ is equal to one then cross-iteration dependencies do not exist between the ith (α-branch) and the jth (β-branch) iterations.

2. Set $k = 1$. Let R_1 be the set of values of the loop index i (which may range in value from N_1 to N_2), for which the loop can be run in parallel in the first "wave". Let $N \equiv \{N_1, N_1+1, N_1+2, \ldots, N_2\}$. The loop index values that belong to R_1 are determined as described by the pseudocode provided in **Table 10** below.

TABLE 10

```
Initialize R1 = {N1}.
do j = N1, N2
  if (C cannot be evaluated now) S(j) = SA(j)
  else {
    if (C) S(j) = ST(j)
    else   S(j) = SF(j)
  }
  if (j = N1) continue;
  do i = N1, j-1
     drop_j = GCD(S(i), S(j)) - 1
     if (drop_j > 0) break          // Indicates that i, j iterations interact.
```

enddo
if (drop_j = 0) $R_1 \leftarrow R_1 \cup \{j\}$
enddo

Following from the pseudocode of **Table 10**, if $R_1 \neq N$, go to step 3, or else go to step 4. The intent of the first loop in **Table 10** is to first check whether the condition in the program loop represented by C in the statement "if (C) ..." can be evaluated before the iteration is executed. For example, a condition appearing in a program loop, such as $C \equiv t(i) - 2 > 0$, where $t(i)$ is an indirect loop index variable, can be evaluated without any of the program loop iterations being executed since the entire t array is known before the loop is entered. On the other hand, a condition such as $C \equiv x2[t(i)]$!= 0 can be evaluated only if $x2[t(i)]$ has not been modified by any previous iteration, otherwise not. If the condition C cannot be evaluated before the program loop iteration is executed, then one cannot *a priori* decide which indirect index variables are actually used during execution and therefore all the indirect index variables in I must be included in the analysis. When the condition C can be evaluated before the program loop iteration is executed, then one of I_T or I_F, as found applicable, is chosen.

3. Set $k \leftarrow k + 1$ for the kth "wave" of parallel computations. Save the loop index values of the kth wave in R_k. To determine the values saved in R_k, proceed as described by the pseudocode provided in **Table 11** below.

TABLE 11

Initialize $R_k = \{l\}$, where l is the smallest index in the set $N - \{R_1 \cup R_2 \cup ... \cup R_{k-1}\}$
do $j = l, N_2$
  if ($j \in R_1 \cup R_2 \cup ... \cup R_{k-1}$) continue
  if (C cannot be evaluated now) $S(j) = S_A(j)$
  else {
    if (C) $S(j) = S_T(j)$
    else $S(j) = S_F(j)$
  }

```
        if (j = l) continue
        do i = l, j-1
           if (i ∈ R1 ∪ R2 ∪ ... ∪ Rk-1) continue
           drop_j = GCD(S(i), S(j)) - 1
           if (drop_j > 0) break
        enddo
        if (drop_j = 0) Rk ← Rk ∪ {j}
     enddo
```

Following from the pseudocode of **Table 11** above, if $R_1 \cup R_2 \cup ... \cup R_k \neq N$, repeat step 3, or else go to step 4.

4. All loop index values saved in a given R_k can be run in the kth "wave". Let n_k be the number of loop index values (n_k is the number of iterations) saved in R_k. Let n_P be the number of available processors over which the iterations can be distributed for parallel execution of the loop. The iterations can be scheduled in many ways, especially if all the processors are not of the same type (for example, in terms of speed, etc). A simple schedule is as follows: Each of the first $n_l = n_k$ mod n_P processors is assigned successive blocks of $(n_k/n_P + 1)$ iterations, and the remaining processors are assigned n_k/n_P iterations.

5. The "waves" are executed one after the other, in sequence, subject to the condition that the next wave cannot commence execution until the previous "wave" completely executes. This is referred to as the wave synchronization criterion.

In relation to the above described procedure of steps 1 to 5, the following observations are made.

(a) In step 1, the $S_\alpha(i)$s, that is, $S_A(i)$, $S_T(i)$, and $S_F(i)$, can be calculated in parallel.

(b) The GCDs of $S(i)$ and $S(j)$ are calculated for $j = N_1+1$ to N_2 and for $i = N_1$ to j-1. The calculations are performed in parallel since each GCD can be calculated independently.

(c) A possible way of parallelizing steps 2 and 3 is to dedicate one processor to these calculations. Let this particular processor calculate R_1. When R_1 is calculated, other processors start calculating the loop iterations according to R_1, while the particular processor starts calculating R_2. When R_2 is calculated and the loop iterations according to R_1 are completed, the other processors start calculating the loop iterations according to R_2, while the same particular processor starts calculating R_3, and so on.

Procedural overview

Before providing example applications of the described techniques, an overview of these described techniques is now provided with reference to **Fig. 1**. **Fig. 1** is a flow chart of steps involved in performing the described techniques. A set of active array variables and a set of indirect loop index variables are determined for the loop under consideration in step **110**. Respective direct loop index values and indirect loop index values are determined in step **120**.

Indirect loop index values $i_1(i)$, $i_2(i)$, ..., $i_r(i)$ are determined for each iteration, in step **130**. Each such value so determined in step **130** is associated with a unique prime number in step **140**. For each iteration, an array of values is then calculated that represents an indirectly indexed access pattern for that iteration, in step **150**.

A grouping of iterations into a minimum number of waves is made such that the iterations comprising a wave are executable in parallel in step **160**.

Finally, the waves are sequentially scheduled in an orderly fashion to allow their respective iterations to execute in parallel in step **170**.

Figs. 2A, **2B** and **2C** present a flow chart of steps that outline, in greater detail, steps involved in performing run-time parallelization as described above. The flow charts are easy to understand if reference is made to **Table 10** for Fig. **2A** and to **Table 11** for **Figs. 2B** and **2C**. Initially, in step **202**, active variables, V, V_T, V_F and their corresponding loop index variables I, I_T, I_F are identified in the loop body. In this notation, the set V is assigned as the union of sets V_T and V_F, and set I is assigned as the union of sets I_T and I_F. Values for N_1, N_2, and M_1 and M_2 are determined, and prime numbers $p(l)$ are assigned to each value of l in the inclusive range defined by $[M_1, M_2]$.

Next, in step **204**, arrays are created as defined in Equation **[1]** below.

$$S_A(i) = \prod_{q \in I} p(q(i))$$

$$S_T(i) = \prod_{q \in I_T} p(q(i))$$

$$S_F(i) = \prod_{q \in I_F} p(q(i)) \qquad [1]$$

Also, k is assigned as 1, the set R_1 is assigned as $\{N_1\}$, and j is assigned as N_1. A determination is then made in step **206** whether the Boolean condition C can be evaluated. If C cannot be evaluated now, $S(j)$ is assigned as $S_A(j)$ in step **208**. Otherwise, if C can be evaluated now, a determination is made in step **210** whether C is *true* or *false*.

If C is *true*, $S(j)$ is assigned as $S_T(j)$ in step **212**. Otherwise, $S(j)$ is assigned as $S_F(j)$ in step **214**. After performing steps **208**, **212** or **214**, a determination is made in step **216** of whether j is equal to N_1, or whether there has been a change in j following step **204**.

If j has changed, then i is assigned as N_1 in step **218**, and drop_j is assigned as the greatest common divisor of $S(i)$ and $S(j)$ less one in step **220**. A determination of whether drop_j is greater than 0 is made in step **222**. If drop_j is *not* greater than 0, then i is incremented by one in step **224**, and a determination is made of whether i is equal to j in step **226**.

If i is *not* equal to j in step **226**, then processing returns to step **220**, in which drop_j is assigned to be the greatest common divisor of $S(i)$ and $S(j)$ less one. Processing proceeds directly to step **222**, as described directly above. If i is equal to j in step **226**, then processing proceeds directly to step **228.**

If drop_j is greater than 0 in step **222**, or if i equals j in step **226**, then a determination is made in step **228** of whether drop_j is equal to 0. If drop_j is equal to 0, the set R_1 is augmented with the set $\{j\}$ by a set union operation. The variable j is then incremented by 1 in step **232**. If drop_j is not equal to 0 in step **228**, then processing proceeds directly to step **232** in which the value of j is incremented by 1.

Once j is incremented in step **232**, a determination is made in step **234** of whether the value of j is greater than the value of N_2. If j is *not* greater than N_2, then processing returns to step **206** to determine whether C can be evaluated, as described above. Otherwise, if j is greater than N_2, a determination is made of whether R_1 is equal to N in step **236**. If R_1 is not equal to N in step **236**, then processing proceeds to step **238**: the value of k is incremented by one, and R_k is assigned as $\{l\}$, where l is the smallest index in the set N less the set formed by the union of sets R_1 through to R_{k-1}. Also, j is assigned to be equal to l.

After this step **238**, a determination is made of whether j is an element of the union of each of the sets R_1 through to R_{k-1}. If j is such an element in step **240**, then j is incremented by one in step **242**. A determination is then made in step **244** of whether the value of j is less than or equal to the value of N_2. If j is indeed less than or equal to the value of N_2 in step **244**, then processing returns to step **240**. Otherwise, processing proceeds to step **278**, as described below, if the value of j is determined to be greater than the value of N_2.

If in step **240**, j is determined to be *not* such an element, then a determination is made in step **246** of whether the Boolean condition C can be evaluated. If C cannot be evaluated in step **246**, then $S(j)$ is assigned as $S_A(j)$.

If, however, C can be evaluated, then in step **250** a determination is made of whether C is *true* or *false*. If C is *true*, $S(j)$ is assigned as $S_T(j)$ in step **252**, otherwise $S(j)$ is assigned as $S_F(j)$ in step **254**.

After performing either of steps **248**, **252**, or **254** as described above, a determination is made in step **256** of whether j is equal to l, namely whether there has been a change in j following step **238**.

If the value of j is *not* equal to l, then the value of i is assigned as l in step **258**. Following step **258**, a determination is made in step **260** of whether i is an element of the union of sets R_1 through to R_{k-1}. If i is *not* an element, then drop_j is assigned to be the greatest common divisor of $S(i)$ and $S(j)$, less one, in step **262**. Then a determination is made in step **264** of whether drop_j is greater than zero. If drop_j is *not* greater than zero, then the value of i is incremented by one in step **266**. Then a determination is made in step **268** of whether the value of i is equal to the value of j in step **268**. If the values of i and j are *not* equal in step **268**, then processing returns to step **260** as described above.

If, however, the values of i and j are equal in step **268**, then a determination is made in step **270** of whether drop_j is equal to zero. If drop_j is equal to zero in step **270**, then the set R_k is augmented by the set $\{j\}$ using a union operator. If drop_j is not equal to zero in step **270**, then the value of j is incremented by one in step **274**. The value of j is also incremented by one in step **274** directly after performing step **272**, or after performing step **256**, if the value of j is found to equal the value of l.

After incrementing the value of j in step **274**, a determination is made in step **276** of whether the value of j is greater than the value of N_2. If the value of j is *not* greater than the value of N_2, then processing returns to step **240**, as described above. Otherwise, if the value of j is greater than the value of N_2, then processing proceeds to step **278**. Step **278** is also performed if the value of j is determined to be greater than N_2 in step **244**, as described above.

In step **278**, a determination is made of whether the set N is equal to the union of sets R_1 through to R_k. If there is no equality between these two sets in step **278**, then processing returns to step **238**, as described above. Otherwise, if the two sets are determined to be equal in step **278**, then step **280** is performed, in which the value of k is saved, and the value of i is assigned as a value of one. Step **280** is also performed following step **236**, if set N is determined to equal set R_1.

Following step **280**, a determination is made in step **282** of whether the value of i is greater than the value of k. If the value of i is greater than the value of k in step **282**, then processing stops in step **286**. Otherwise, if the value of i is less than or equal to the value of k in step **282**, then step **284** is performed in which iterations are executed in parallel for loop index values that are saved in the set R_i. The value of i is also incremented by one, and processing then returns to step **282** as described above.

Example 1

A first example is described with reference to the code fragment of **Table 12** below.

TABLE 12

```
do i = 5, 9
  x1[t(i)] = x2[r(i)]
  if (t(i) > 2) x2[u(i)] = x1[v(i)]
  else          x2[u(i)] = x1[t(i)]
enddo
```

In **Table 12** above, since **x1** and **x2** are the only active array variables, the indirect loop index variables **r(i)**, **t(i)**, **u(i)**, **v(i)** associated with these variables are the only index variables that are considered. The values of **r(i)**, **t(i)**, **u(i)**, **v(i)** are provided in **Table 13** below.

TABLE 13

Indirect index variable	i = 5	i = 6	i = 7	i = 8	i = 9
r(i)	1	2	3	4	4
t(i)	1	2	2	1	4
u(i)	1	2	2	4	1
v(i)	1	2	3	1	1

By inspection, $M_1 = 1$, $M_2 = 4$, and $N_1 = 5$, $N_2 = 9$. A unique prime number is associated with each of the values 1, 2, 3, 4 that one or more of the indirect index variables can attain: $p(1) = 3$, $p(2) = 5$, $p(3) = 7$, $p(4) = 11$.

The pseudocode in **Table 14** below illustrates the operations that are performed with reference to steps 1 to 5 described above in the subsection entitled "*Loop parallelization procedure*".

TABLE 14

Step 1

$S_A(i) = S_T(i) = p(r(i)) \times p(t(i)) \times p(u(i)) \times p(v(i))$ for $i = 5, 6, 7, 8, 9$.

$S_A(5) = S_T(5) = p(1) \times p(1) \times p(1) \times p(1) = 3 \times 3 \times 3 \times 3 = 81$

$S_A(6) = S_T(6) = p(2) \times p(2) \times p(2) \times p(2) = 5 \times 5 \times 5 \times 5 = 625$

$S_A(7) = S_T(7) = p(3) \times p(2) \times p(2) \times p(3) = 7 \times 5 \times 5 \times 7 = 1225$

$S_A(8) = S_T(8) = p(4) \times p(1) \times p(4) \times p(1) = 11 \times 3 \times 11 \times 3 = 1089$

$S_A(9) = S_T(9) = p(4) \times p(4) \times p(1) \times p(1) = 11 \times 11 \times 3 \times 3 = 1089$

$S_F(i) = p(r(i)) \times p(t(i)) \times p(u(i))$ for $i = 5, 6, 7, 8, 9$.

$S_F(5) = p(1) \times p(1) \times p(1) = 3 \times 3 \times 3 = 27$

$S_F(6) = p(2) \times p(2) \times p(2) = 5 \times 5 \times 5 = 125$

$S_F(7) = p(3) \times p(2) \times p(2) = 7 \times 5 \times 5 = 175$

$S_F(8) = p(4) \times p(1) \times p(4) = 11 \times 3 \times 11 = 363$

$S_F(9) = p(4) \times p(4) \times p(1) = 11 \times 11 \times 3 = 363$

Step 2

Set $k = 1$, $R_1 = \{5\}$.

$j = 5$:

if cond = FALSE; $S(5) = S_F(5) = 27$;

$j = 6$:

if cond = FALSE; $S(6) = S_F(6) = 125$;

$i = 5$: GCD(27, 125) = 1;

$R_1 = \{5, 6\}$

$j = 7$:

if cond = FALSE; $S(7) = S_F(7) = 175$;

$i = 5$: GCD(27, 175) = 1;

$i = 6$: GCD(125, 175) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

$j = 8$:

if cond = FALSE; $S(8) = S_F(8) = 363$;

$i = 5$: GCD(27, 363) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

$j = 9$:

if cond = TRUE; $S(9) = S_T(9) = 1089$;

$i = 5$: GCD(27, 1089) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

Since $R_1 \neq N$, go to step 3.

Step 3

Set $k = 2$, $l = 7$, $R_2 = \{7\}$.

$j = 7$:

$j \notin R_1$;

if cond = FALSE; $S(7) = S_F(7) = 175$;

$j = 8$:

$j \notin R_1$;

if cond = FALSE; $S(8) = S_F(8) = 363$;

$i = 7$: $i \notin R_1$; GCD(175, 363) = 1;

$R_2 = \{7, 8\}$

$j = 9$:

$j \notin R_1$;

if cond = TRUE; $S(9) = S_T(9) = 1089$;

$i = 7$: $i \notin R_1$; GCD(175, 1089) = 1;

$i = 8$: $i \notin R_1$; GCD(363, 1089) $\neq$ 1; terminate loop

$R_2 = \{7, 8\}$

Since $R_1 \cup R_2 \neq N$, repeat step 3.

Set $k = 3$, $l = 9$, $R_3 = \{9\}$.

$j = 9$:

$j \notin (R_1 \cup R_2)$;

if cond = TRUE; $S(9) = S_T(9) = 1089$;

No further iterations.

$R_3 = \{9\}$

Since $R_1 \cup R_2 \cup R_3 = N$, go to step 4.

Steps 4 and 5

Execute as outlined in steps 4 and 5 in the subsection entitled "*Loop parallelization procedure*". Notice that there are 5 iterations and 3 waves: $R_1 = \{5, 6\}$, $R_2 = \{7, 8\}$, $R_3 = \{9\}$.

Example 2

A second example is described with reference to the code fragment of **Table 15** below.

TABLE 15

```
do i = 5, 9
  x1[t(i)] = x2[r(i)] + . . .
  if (x1[t(i)] > 0) x2[u(i)] = x1[v(i)] + . . .
  else              x2[u(i)] = x1[t(i)] + . . .
enddo
```

In the example of **Table 15** above, since **x1**, **x2** are the only active array variables, the indirect loop index variables **r(i)**, **t(i)**, **u(i)**, **v(i)** associated with these variables are the index variables that are considered for parallelization. Values of **r(i)**, **t(i)**, **u(i)**, **v(i)** are tabulated in **Table 16** below.

TABLE 16

Indirect index variable	i = 5	i = 6	i = 7	i = 8	i = 9
r(i)	1	2	3	4	4
t(i)	1	2	2	1	4
u(i)	1	2	2	4	1
v(i)	1	2	3	3	1

By inspection, $M_1 = 1$, $M_2 = 4$, and $N_1 = 5$, $N_2 = 9$. A unique prime number is associated with each of the values 1, 2, 3, 4 that one or more of the indirect index variables attains: $p(1) = 3$, $p(2) = 5$, $p(3) = 7$, $p(4) = 11$. That is, $p()$ simply provides consecutive prime numbers, though any alternative sequence of prime numbers can also be used.

The pseudocode in **Table 17** below illustrates the operations that are performed with reference to steps 1 to 5 described above in the subsection entitled "*Loop parallelization procedure*".

TABLE 17

Step 1

$S_A(i) = S_T(i) = p(r(i)) \times p(t(i)) \times p(u(i)) \times p(v(i))$ for $i = 5, 6, 7, 8, 9$.

$S_A(5) = S_T(5) = p(1) \times p(1) \times p(1) \times p(1) = 3 \times 3 \times 3 \times 3 = 81$

$S_A(6) = S_T(6) = p(2) \times p(2) \times p(2) \times p(2) = 5 \times 5 \times 5 \times 5 = 625$

$S_A(7) = S_T(7) = p(3) \times p(2) \times p(2) \times p(3) = 7 \times 5 \times 5 \times 7 = 1225$

$S_A(8) = S_T(8) = p(4) \times p(1) \times p(4) \times p(3) = 11 \times 3 \times 11 \times 7 = 2541$

$S_A(9) = S_T(9) = p(4) \times p(4) \times p(1) \times p(1) = 11 \times 11 \times 3 \times 3 = 1089$

$S_F(i) = p(r(i)) \times p(t(i)) \times p(u(i))$ for $i = 5, 6, 7, 8, 9$.

$S_F(5) = p(1) \times p(1) \times p(1) = 3 \times 3 \times 3 = 27$

$S_F(6) = p(2) \times p(2) \times p(2) = 5 \times 5 \times 5 = 125$

$S_F(7) = p(3) \times p(2) \times p(2) = 7 \times 5 \times 5 = 175$

$S_F(8) = p(4) \times p(1) \times p(4) = 11 \times 3 \times 11 = 363$

$S_F(9) = p(4) \times p(4) \times p(1) = 11 \times 11 \times 3 = 363$

Step 2

Set $k = 1$, $R_1 = \{5\}$.

$j = 5$:

if cond cannot be evaluated; $S(5) = S_A(5) = 81$;

$j = 6$:

if cond cannot be evaluated; $S(6) = S_A(6) = 625$;

$i = 5$: GCD(81, 625) = 1;

$R_1 = \{5, 6\}$

$j = 7$:

if cond cannot be evaluated; $S(7) = S_A(7) = 1225$;

$i = 5$: GCD(81, 1225) = 1;

$i = 6$: GCD(625, 1225) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

$j = 8$:

if cond cannot be evaluated; $S(8) = S_A(8) = 2541$;

$i = 5$: GCD(81, 2541) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

$j = 9$:

if cond cannot be evaluated; $S(9) = S_A(9) = 1089$;

$i = 5$: GCD(81, 1089) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

Since $R_1 \neq N$, go to step 3.

Step 3

Set $k = 2$, $l = 7$, $R_2 = \{7\}$.

$j = 7$:

$j \notin R_1$;

if cond cannot be evaluated; $S(7) = S_A(7) = 1225$;

$j = 8$:

$j \notin R_1$;

if cond cannot be evaluated; $S(8) = S_A(8) = 2541$;

$i = 7$: $i \notin R_1$; GCD(1225, 2541) $\neq 1$; terminate loop

$R_2 = \{7\}$

$j = 9$:

$j \notin R_1$;

if cond cannot be evaluated; $S(9) = S_A(9) = 1089$;

$i = 7$: $i \notin R_1$; GCD(1225, 1089) = 1;

$i = 8$: $i \notin R_1$; GCD(2541, 1089) $\neq 1$; terminate loop

$R_2 = \{7\}$

Since $R_1 \cup R_2 \neq N$, repeat step 3.

Set $k = 3$, $l = 8$, $R_3 = \{8\}$.

$j = 8$:

$j \notin (R_1 \cup R_2)$;

if cond cannot be evaluated; $S(8) = S_A(8) = 2541$;

$j = 9$:

$j \notin (R_1 \cup R_2)$;

if cond cannot be evaluated; $S(9) = S_A(9) = 1089$;

$i = 8$: $i \notin (R_1 \cup R_2)$; GCD(2541, 1089) $\neq 1$; terminate loop

$R_3 = \{8\}$

Set $k = 4$, $l = 9$, $R_4 = \{9\}$.

$j = 9$:

$j \notin (R_1 \cup R_2 \cup R_3)$;

if cond cannot be evaluated; $S(9) = S_A(9) = 1089$;

No further iterations.

$R_4 = \{9\}$

Since $R_1 \cup R_2 \cup R_3 \cup R_4 = N$, go to step 4.

Steps 4 and 5

Execute as outlined in steps 4 and 5 in the subsection entitled "*Loop parallelization procedure*". Notice that in this example there are 5 iterations and 4 waves: $R_1 = \{5, 6\}$, $R_2 = \{7\}$, $R_3 = \{8\}$, $R_4 = \{9\}$.

Example **3**

A third example is described with reference to the code fragment of **Table 18** below.

TABLE 18

```
do i = 5, 9
  x1[t(i)] = x2[r(i)] + . . .
  if (x1[t(i)] > 0 || t(i) > 2) x2[u(i)] = x1[v(i)] + . . .
  else                          x2[u(i)] = x1[t(i)] + . . .
enddo
```

In the example of **Table 18** above, since **x1**, **x2** are the only active array variables, the indirect loop index variables **r(i)**, **t(i)**, **u(i)**, **v(i)** associated with them are the index variables to be considered for parallelization.

Values of **r(i)**, **t(i)**, **u(i)**, and **v(i)** are tabulated in **Table 19** below.

TABLE 19

Indirect index variable	i = 5	i = 6	i = 7	i = 8	i = 9
r(i)	1	2	3	4	4
t(i)	1	2	3	1	4
u(i)	1	2	2	4	1
v(i)	1	2	3	3	1

By inspection, $M_1 = 1$, $M_2 = 4$, and $N_1 = 5$, $N_2 = 9$. A unique prime number is associated with each of the values 1, 2, 3, 4 that one or more of the indirect index variables attains: $p(1) = 3, p(2) = 5, p(3) = 7, p(4) = 11$.

The pseudocode in **Table 20** below illustrates the operations that are performed with reference to steps 1 to 5 described above in the subsection entitled "*Loop parallelization procedure*".

TABLE 20

Step 1

$S_A(i) = S_T(i) = p(r(i)) \times p(t(i)) \times p(u(i)) \times p(v(i))$ for $i = 5, 6, 7, 8, 9$.

$S_A(5) = S_T(5) = p(1) \times p(1) \times p(1) \times p(1) = 3 \times 3 \times 3 \times 3 = 81$

$S_A(6) = S_T(6) = p(2) \times p(2) \times p(2) \times p(2) = 5 \times 5 \times 5 \times 5 = 625$

$S_A(7) = S_T(7) = p(3) \times p(3) \times p(2) \times p(3) = 7 \times 7 \times 5 \times 7 = 1715$

$S_A(8) = S_T(8) = p(4) \times p(1) \times p(4) \times p(3) = 11 \times 3 \times 11 \times 7 = 2541$

$S_A(9) = S_T(9) = p(4) \times p(4) \times p(1) \times p(1) = 11 \times 11 \times 3 \times 3 = 1089$

$S_F(i) = p(r(i)) \times p(t(i)) \times p(u(i))$ for $i = 5, 6, 7, 8, 9$.

$S_F(5) = p(1) \times p(1) \times p(1) = 3 \times 3 \times 3 = 27$

$S_F(6) = p(2) \times p(2) \times p(2) = 5 \times 5 \times 5 = 125$

$S_F(7) = p(3) \times p(3) \times p(2) = 7 \times 7 \times 5 = 245$

$S_F(8) = p(4) \times p(1) \times p(4) = 11 \times 3 \times 11 = 363$

$S_F(9) = p(4) \times p(4) \times p(1) = 11 \times 11 \times 3 = 363$

Step 2

Set $k = 1$, $R_1 = \{5\}$.

$j = 5$:

' if cond' cannot be evaluated; $S(5) = S_A(5) = 81$;

Comment: The 'if cond' cannot be evaluated since even though 't(i) > 2' is false, the 'or' operator requires that x1[t(i)] must also be evaluated to finally determine the 'if cond'. If the 'if cond' had turned out to be true, then evaluation of x1[t(i)] would not have been necessary in view of the 'or' operator.

$j = 6$:

' if cond' cannot be evaluated; $S(6) = S_A(6) = 625$;

$i = 5$: GCD(81, 625) = 1;

$R_1 = \{5, 6\}$

$j = 7$:

if cond = TRUE; $S(7) = S_T(7) = 1715$;

Comment: The 'if cond' is true because 't(i) > 2' is true. Therefore x1[t(i)] need not be evaluated in the presence of the 'or' operator.

$i = 5$: GCD(81, 1715) = 1;

$i = 6$: GCD(625, 1715) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

$j = 8$:

'if cond' cannot be evaluated; $S(8) = S_A(8) = 2541$;

$i = 5$: GCD(81, 2541) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

$j = 9$:

'if cond' = TRUE; $S(9) = S_T(9) = 1089$;

Comment: The 'if cond' is true because 't(i) > 2' is true. Therefore x1[t(i)] need not be evaluated in the presence of the 'or' operator.

$i = 5$: GCD(81, 1089) $\neq$ 1; terminate loop

$R_1 = \{5, 6\}$

Since $R_1 \neq N$, go to step 3.

Step 3

Set $k = 2$, $l = 7$, $R_2 = \{7\}$.

$j = 7$:

$j \notin R_1$;

'if cond' = TRUE; $S(7) = S_T(7) = 1715$;

$j = 8$:

$j \notin R_1$;

'if cond' cannot be evaluated; $S(8) = S_A(8) = 2541$;

$i = 7$: $i \notin R_1$; GCD(1715, 2541) $\neq 1$; terminate loop

$R_2 = \{7\}$

$j = 9$:

$j \notin R_1$;

' if cond' = TRUE; $S(9) = S_T(9) = 1089$;

$i = 7$: $i \notin R_1$; GCD(1715, 1089) = 1;

$i = 8$: $i \notin R_1$; GCD(2541, 1089) $\neq 1$; terminate loop

$R_2 = \{7\}$

Since $R_1 \cup R_2 \neq N$, repeat step 3.

Set $k = 3$, $l = 8$, $R_3 = \{8\}$.

$j = 8$:

$j \notin (R_1 \cup R_2)$;

'if cond' cannot be evaluated; $S(8) = S_A(8) = 2541$;

$j = 9$:

$j \notin (R_1 \cup R_2)$;

'if cond' = TRUE; $S(9) = S_T(9) = 1089$;

$i = 8$: $i \notin (R_1 \cup R_2)$; GCD(2541, 1089) $\neq 1$; terminate loop

$R_3 = \{8\}$

Set $k = 4$, $l = 9$, $R_4 = \{9\}$.

$j = 9$:

$j \notin (R_1 \cup R_2 \cup R_3)$;

'if cond' = TRUE; $S(9) = S_T(9) = 1089$;

No further iterations.

$R_4 = \{9\}$

Since $R_1 \cup R_2 \cup R_3 \cup R_4 = N$, go to step 4.

Steps 4 and 5

Execute as outlined in steps 4 and 5 in the subsection entitled "*Loop parallelization procedure*". Notice that in this example too there are 5 iterations and 4 waves: $R_1 = \{5, 6\}$, $R_2 = \{7\}$, $R_3 = \{8\}$, $R_4 = \{9\}$.

Case when no conditional statements are present in the loop

In this case put $V = V_A$, $I = I_A$, $S = S_A$. Since there is no conditional statement C in the loop, the statement "if (C cannot be evaluated now) ...", wherever it appears in the loop parallelization algorithm described above, is assumed to evaluate to "true".

Extension of the method to include multiple Boolean conditions

Inclusion of more than one Boolean condition in a loop body increases the number of decision paths (to a maximum of 3^r, where r is the number of Boolean conditions) available in a loop. The factor 3 appears because each condition may have one of three states: *true*, *false*, *not decidable*, even though the condition is Boolean. For each path λ, it is necessary to compute an $S_\lambda(i)$ value for each iteration i. This is done by modifying the code fragment shown in **Table 21** which appears in steps 2 and 3 of the "*Loop parallelization procedure*" described above.

TABLE 21

```
if (C is not d cidable) S(j) = SA(j)
els  {
```

```
    if (C) S(j) = S_T(j)
    else   S(j) = S_F(j)
}
```

The modification replaces the code fragment by

$$\text{if } (\lambda = \text{path}(i))\ S(i) = S_\lambda(i)$$

where the function *path*(*i*) evaluates the Boolean conditions in the path and returns a path index λ. The enumeration of all possible paths, for each loop in a program, can be done by a compiler and the information provided to the run-time system in an appropriate format. Typically, each Boolean condition is provided with a unique identifier, which is then used in constructing the paths. When such an identifier appears in a path it is also tagged with one of three states, say, *T* (for *true*), *F* (for *false*), *A* (for *not decidable*, that is, carry *all* active array variables) as applicable for the path. A suggested path format is the following string representation

***ident_1*:X_1 *ident_2*:X_2 . . . *ident_n*:X_n;**,

where ***ident_i*** identifies a Boolean condition in a loop and ***X_i*** one of its possible state *T*, *F*, or *A*. Finally, this string is appended with the list of indirect loop index variables that appear with the active variables in the path. A suggested format is

***ident_1*:X_1 *ident_2*:X_2 . . . *ident_n*:X_n; {I_λ}**,

where **{I_λ}** comprises the set of indirect loop index variables (any two variables being separated by a comma), and the construction of any of ***ident_n***, ***X_n***, or elements of the set **{I_λ}** do not use the delimiter characters '**:**', '**;**' or '**,**'. The left-to-right sequence in which the identifiers appear in a path string corresponds to the sequence in which the Boolean conditions will be encountered in the path at run-time. Let $Q = \{q_1, q_2, \ldots, q_m\}$ be the set of *m* appended path strings found by a compiler. A typical appended path string q_λ in *Q* may appear as

$q_\lambda \equiv$ **id4:T id7:T id6:F id8:T; {u, r, t}**,

where the path portion represents the execution sequence wherein the Boolean condition with the identifier **id4** evaluates to *true*, **id7** evaluates to *true*, **id6** evaluates to *false*, **id8** evaluates to *true*, and the path has the indirect loop index variables **{u, r, t}** associated with its active variables.

With the formatted set Q of all possible appended path strings available from a compiler, the run-time system then needs only to construct a path q for each iteration being considered in a wave, compare q with the paths in Q, and decide upon the parallelizing options available to it.

The simplest type of path the run-time system can construct is one for which each Boolean condition, in the sequence of Boolean conditions being evaluated in an iteration, evaluates to either *true* or *false*. In such a case, the exact path in the iteration is known. Let q be such a path, which in the suggested format appears as

$q \equiv$ **ident_1:X_1 ident_2:X_2 . . . ident_n:X_n;**.

A string match with the set of strings available in Q will show that q will appear as a path in one and only one of the strings in Q (since q was cleverly formatted to end with the character '**;**' which does not appear in any other part of the string), say, q_λ and the function *path*(i) will return the index λ on finding this match. The set of indirect loop index variables $\{I_\lambda\}$ can be plucked from the trailing part of q_λ for calculating $S_\lambda(i)$.

When the run-time system, while constructing a path q, comes across a Boolean condition that evaluates to *not decidable*, it means that a definite path cannot be determined before executing the iteration. In such a case, the construction of the path is terminated at the undecidable Boolean condition encountered after encoding the Boolean condition and its state (A) into the path string. For example, let this undecidable Boolean condition have the identifier **idr**, then the path q would terminate with the substring **idr:A;**. A variation of q is now constructed which is identical to q except that the character '**;**' is replaced by the blank character ' '. Let q' be this variation. All the strings in Q for which

either q or q' is an initial substring (meaning that q will appear as a substring from the head of whatever string in Q it matches with) is a possible path for the iteration under consideration. (There will be more than one such path found in Q.) In such a case the *path*() function will return an illegal λ value (in this embodiment it is -1) and $S_\lambda(i)$ is computed using the set of indirect index variables given by the union of all the indirect index variable sets that appear in the paths in Q for which either of q or q' was found to be an initial substring. Note that $S_{-1}(i)$ does not have a unique value (unlike the other $S_\lambda(i)$s which could be precalculated and saved) but must be calculated afresh every time *path*(i) returns -1.

Nested indexing of indirect index variables

The case in which one or more of the indirect index variables, for example, i_k, is further indirectly indexed as $i_k(l)$ where $l(i)$, in turn, is indirectly indexed to i, is handled by treating $i_k(l)$ as another indirect index variable, for example, $i_t(i)$. Indeed, l, instead of being an array can be any function of i.

Use of bit vectors instead of prime numbers

Instead of defining $S_\lambda(i)$, where λ is a decision path in the loop, in terms of the product of prime numbers, one may use a binary bit vector. Here one associates a binary bit, in place of a prime number, for each number in the range $[M_1, M_2]$. That is, the k-th bit of a bit vector $S_\lambda(i)$ when set to 1 denotes the presence of the prime number $p(k)$ in $S_\lambda(i)$. Alternatively, the notation $b_{\lambda i}$ may be used for the k-th bit of this bit vector. If a logical AND operation between any two bit vectors $S_\alpha(i)$ and $S_\beta(j)$ produces a null bit vector, then the decision paths corresponding to $S_\alpha(i)$ and $S_\beta(j)$ do not share common values of the indirect index variables. This is equivalent to the expression GCD($S_\alpha(i)$, $S_\beta(j)$) = 1 described above.

Computer hardware and software

Fig. 3 is a schematic representation of a computer system **300** that is provided for executing computer software programmed to assist in performing run-time parallelization

of loops as described herein. This computer software executes on the computer system **300** under a suitable operating system installed on the computer system **300**.

The computer software is based upon computer program comprising a set of programmed instructions that are able to be interpreted by the computer system **300** for instructing the computer system **300** to perform predetermined functions specified by those instructions. The computer program can be an expression recorded in any suitable programming language comprising a set of instructions intended to cause a suitable computer system to perform particular functions, either directly or after conversion to another programming language.

The computer software is programmed using statements in an appropriate computer programming language. The computer program is processed, using a compiler, into computer software that has a binary format suitable for execution by the operating system. The computer software is programmed in a manner that involves various software components, or code means, that perform particular steps in accordance with the techniques described herein.

The components of the computer system **300** include: a computer **320**, input devices **310**, **315** and video display **390**. The computer **320** includes: processor **340**, memory module **350**, input/output (I/O) interfaces **360**, **365**, video interface **345**, and storage device **355**. The computer system **300** can be connected to one or more other similar computers, using a input/output (I/O) interface **365**, via a communication channel **385** to a network **380**, represented as the Internet.

The processor **340** is a central processing unit (CPU) that executes the operating system and the computer software executing under the operating system. The memory module **350** includes random access memory (RAM) and read-only memory (ROM), and is used under direction of the processor **340**.

The video interface **345** is connected to video display **390** and provides video signals for display on the video display **390**. User input to operate the computer **320** is provided from input devices **310**, **315** consisting of keyboard **310** and mouse **315**. The storage device **355** can include a disk drive or any other suitable non-volatile storage medium.

Each of the components of the computer **320** is connected to a bus **330** that includes data, address, and control buses, to allow these components to communicate with each other via the bus **330**.

The computer software can be provided as a computer program product recorded on a portable storage medium. In this case, the computer software is accessed by the computer system **300** from the storage device **355**. Alternatively, the computer software can be accessed directly from the network **380** by the computer **320**. In either case, a user can interact with the computer system **300** using the keyboard **310** and mouse **315** to operate the computer software executing on the computer **320**.

The computer system **300** is described only as an example for illustrative purposes. Other configurations or types of computer systems can be equally well used to implement the described techniques.

Various alterations and modifications can be made to the techniques and arrangements described herein, as would be apparent to one skilled in the relevant art.

Conclusion

Techniques and arrangements are described herein for performing run-time parallelization of loops in computer programs having indirect loop index variables and embedded conditional variables. Various alterations and modifications can be made to the techniques and arrangements described herein, as would be apparent to one skilled in the relevant art.